**WEEK 1**

**DESIGN PRINCIPLES AND PATTERNS HANDS ON**

**Excercise 1: Implementing the Singleton Pattern**

**CODE** :

// Excercise 1 : Implementing Singleton Class

// class named logger

public class Logger {

    //  private static instance

    private static Logger instance;

    // private constructor for logger

    private Logger(){

        System.out.println("Logger instance is created");

    }

    // public staic method for getting instance

    public static Logger getInstance(){

        if(instance==null){

            instance = new Logger();

        }

        return instance;

    }

    // method for logging message

    public void log(String message){

        System.out.println("[log]: "+message);

    }

}

//test class to verify only 1 instance of class is created

public class Main {

    public static void main(String[] args){

        // getting the instance for 1st time

        Logger log1 = Logger.getInstance();

        log1.log("This is 1st log message");

        // 2nd time

        Logger log2 = Logger.getInstance();

        log2.log("This is 2nd log message");

        // verifying

        if (log1==log2){

            System.out.println("log1 and log2 are Same Instance");

        }

        else{

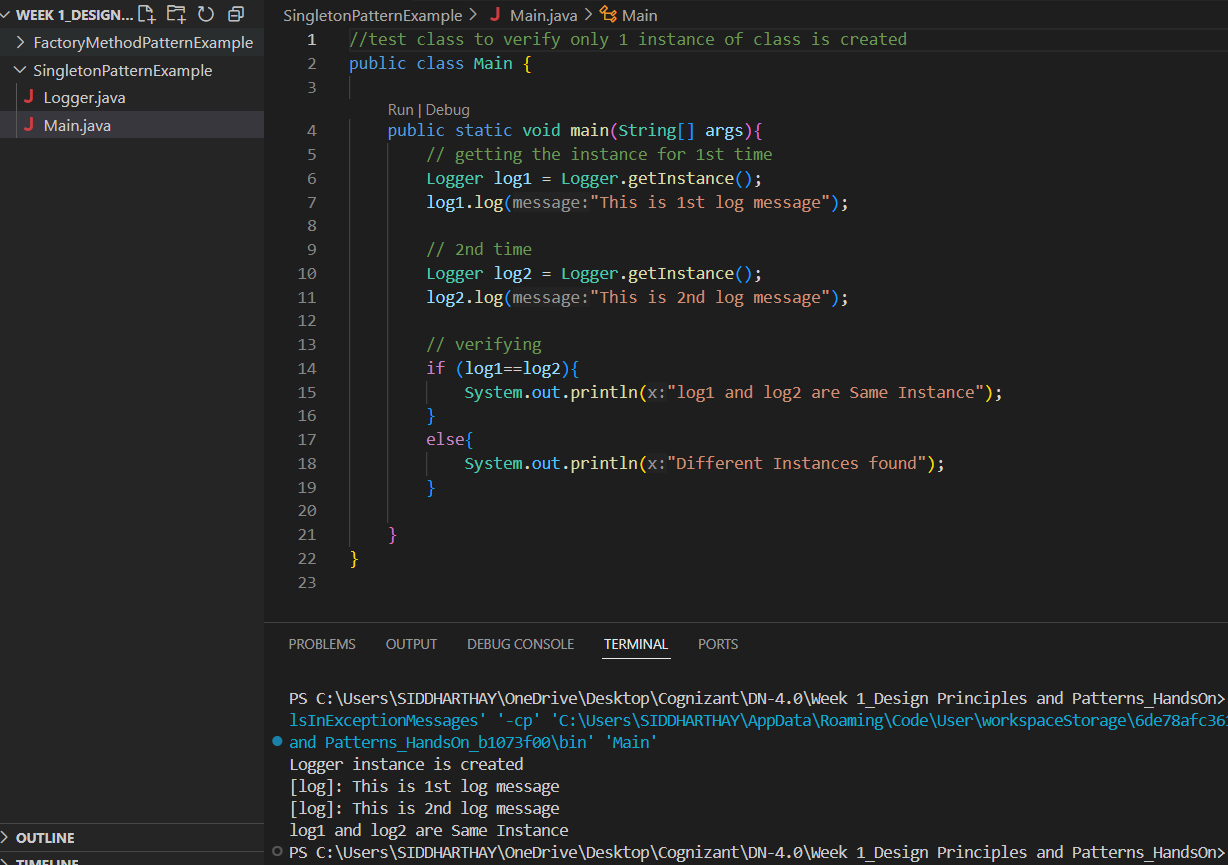
            System.out.println("Different Instances found");

        }

    }

}

**OUTPUT**



**Exercise 2: Implementing the Factory Method Pattern**

**CODE :**

// Document.java

package FactoryMethodPatternExample;

public interface Document{

    void open();

}

public class WordDocument  implements Document{

    public void open(){

        System.out.println("opening word doc");

    }

}

public class PdfDocument implements Document{

    public void open(){

        System.out.println("opening pdf doc");

    }

}

public class ExcelDocument implements Document{

    public void open(){

        System.out.println("opening excel doc");

    }

}

public abstract class DocumentFactory {

    public abstract Document createDoc();

}

public class WordDocumentFactory extends DocumentFactory{

    public Document createDoc(){

        return new WordDocument();

    }

}

public class PdfDocumentFactory extends DocumentFactory{

    public Document createDoc(){

        return new PdfDocument();

    }

}

public class ExcelDocumentFactory  extends DocumentFactory{

    public Document createDoc(){

        return new ExcelDocument();

    }

}

public class Main {

    public static void main(String[] args){

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDoc();

        wordDoc.open();

        DocumentFactory pdfFactory = new WordDocumentFactory();

        Document pdfDoc = pdfFactory.createDoc();

        pdfDoc.open();

        DocumentFactory excelFactory = new WordDocumentFactory();

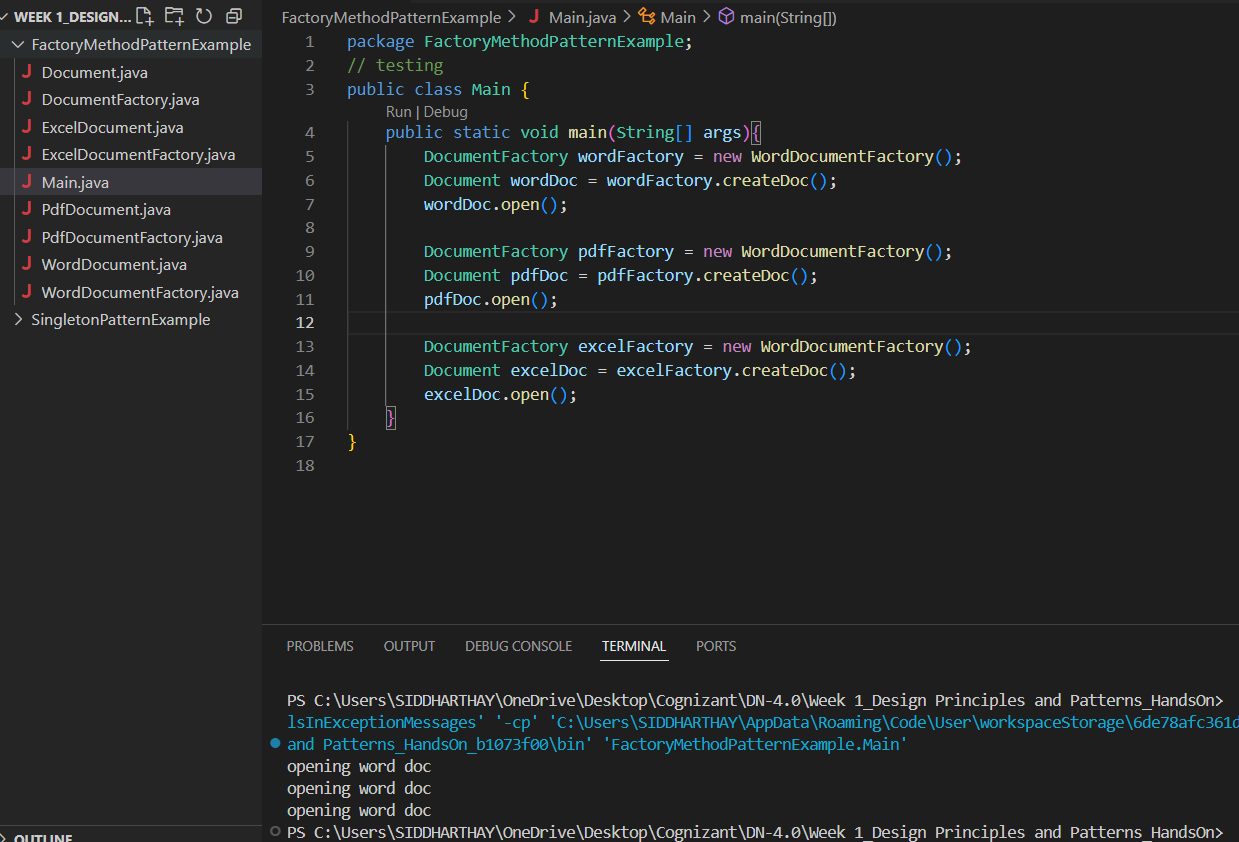
        Document excelDoc = excelFactory.createDoc();

        excelDoc.open();

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**CODE:**

package BuilderPatternExample;

//computer class

public class Computer {

    //attributes

    private String CPU;

    private String RAM;

    private String storage;

    private boolean hasMSoffice;

    private boolean hasGraphicsCard;

    //privateconstructor accessed through builder

    private Computer(Builder builder){

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.hasMSoffice = builder.hasMSoffice;

        this.hasGraphicsCard = builder.hasGraphicsCard;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private boolean hasMSoffice;

        private boolean hasGraphicsCard;

        public Builder setCPU(String CPU) {

            this.CPU = CPU;

            return this;

        }

        public Builder setRAM(String RAM) {

            this.RAM = RAM;

            return this;

        }

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        public Builder setMSoffice(boolean hasMSoffice) {

            this.hasMSoffice = hasMSoffice;

            return this;

        }

        public Builder setGraphicsCard(boolean hasGraphicsCard) {

            this.hasGraphicsCard = hasGraphicsCard;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    //display method to show computer details

    public void displayConfiguration() {

        System.out.println("Computer Configuration:  ");

        System.out.println("CPU: " + CPU);

        System.out.println("RAM: " + RAM);

        System.out.println("Storage: " + storage);

        System.out.println("WiFi: " + (hasMSoffice ? "Yes" : "No"));

        System.out.println("Graphics Card: " + (hasGraphicsCard ? "Yes" : "No"));

    }

}

**//**main class

public class Main {

    public static void main(String[] args) {

        //gaming pc config

        Computer gamingPC = new Computer.Builder()

                .setCPU("Intel Core i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setMSoffice(false)

                .setGraphicsCard(true)

                .build();

        gamingPC.displayConfiguration();

        //normal pc config

        Computer officePC = new Computer.Builder()

                .setCPU("Intel Core i5")

                .setRAM("8GB")

                .setStorage("512GB HDD")

                .setMSoffice(true)

                .setGraphicsCard(false)

                .build();

        officePC.displayConfiguration();

        //budget pc config

        Computer barebonePC = new Computer.Builder()

                .setCPU("Intel Core i3")

                .setRAM("4GB")

                .setStorage("256GB SSD")

                .build();

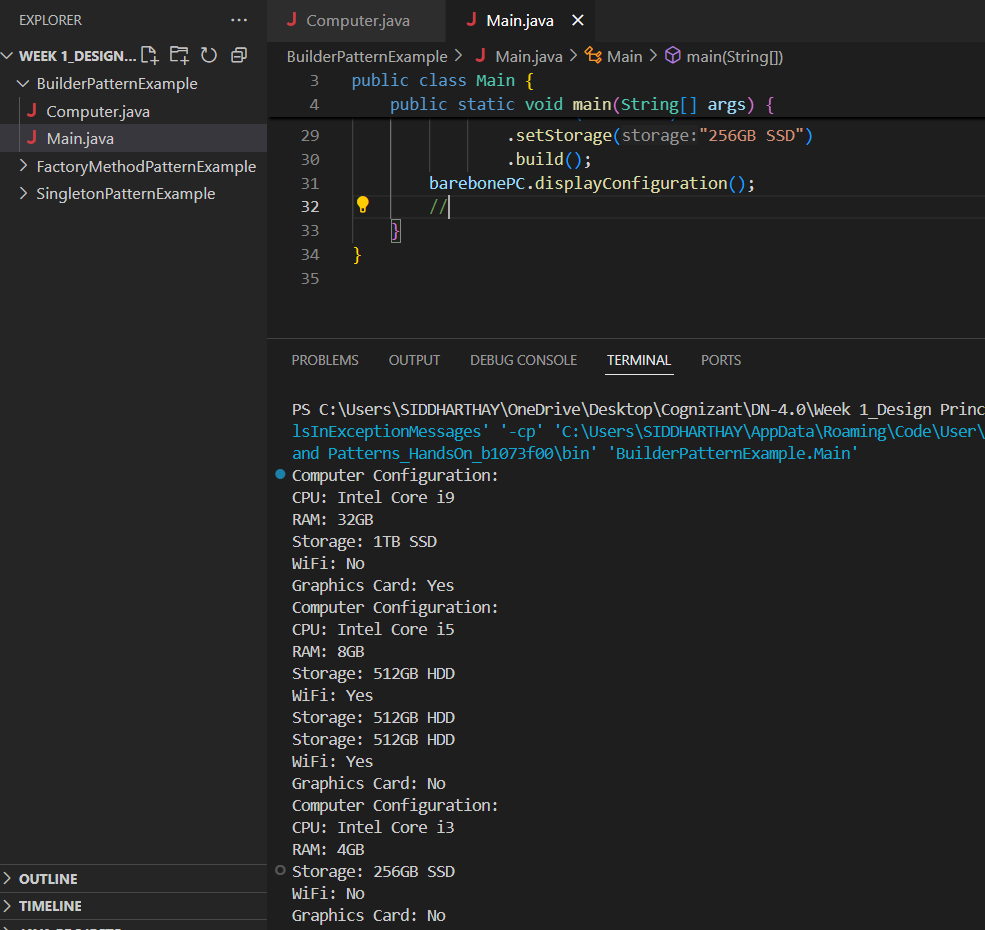
        barebonePC.displayConfiguration();

        //

    }

}

**OUTPUT :**

****

**Exercise 4: Implementing the Adapter Pattern**

**CODE:**

package AdapterPatternExample;

public interface PaymentProcessor {

    void processPayment(double amount);

}

public class PhonepeGateway {

    public void SendMoneyViaPhonepe(double amount) {

        System.out.println("PhonePe processed payment of " + amount);

    }

}

public class GooglepayGateway {

    public void sendMoneyViaGooglepay(double amount) {

        System.out.println("GooglePay processed payment of " + amount);

    }

}

public class PhonepeAdapter implements PaymentProcessor {

    private PhonepeGateway phonePe;

    public PhonepeAdapter(PhonepeGateway phonePe) {

        this.phonePe = phonePe;

    }

    public void processPayment(double amount) {

        //adapts to phonepe method

        phonePe.SendMoneyViaPhonepe(amount);

    }

}

public class GooglepayAdapter implements PaymentProcessor {

    private GooglepayGateway googlePay;

    public GooglepayAdapter(GooglepayGateway googlePay) {

        this.googlePay = googlePay;

    }

    public void processPayment(double amount) {

        //adapts to googlepay method

        googlePay.sendMoneyViaGooglepay(amount);

    }

}

public class Main {

    public static void main(String[] args) {

        //phonepe payment

        PaymentProcessor phonePeProcessor = new PhonepeAdapter(new PhonepeGateway());

        phonePeProcessor.processPayment(7777.00);

        //googlepay paymen

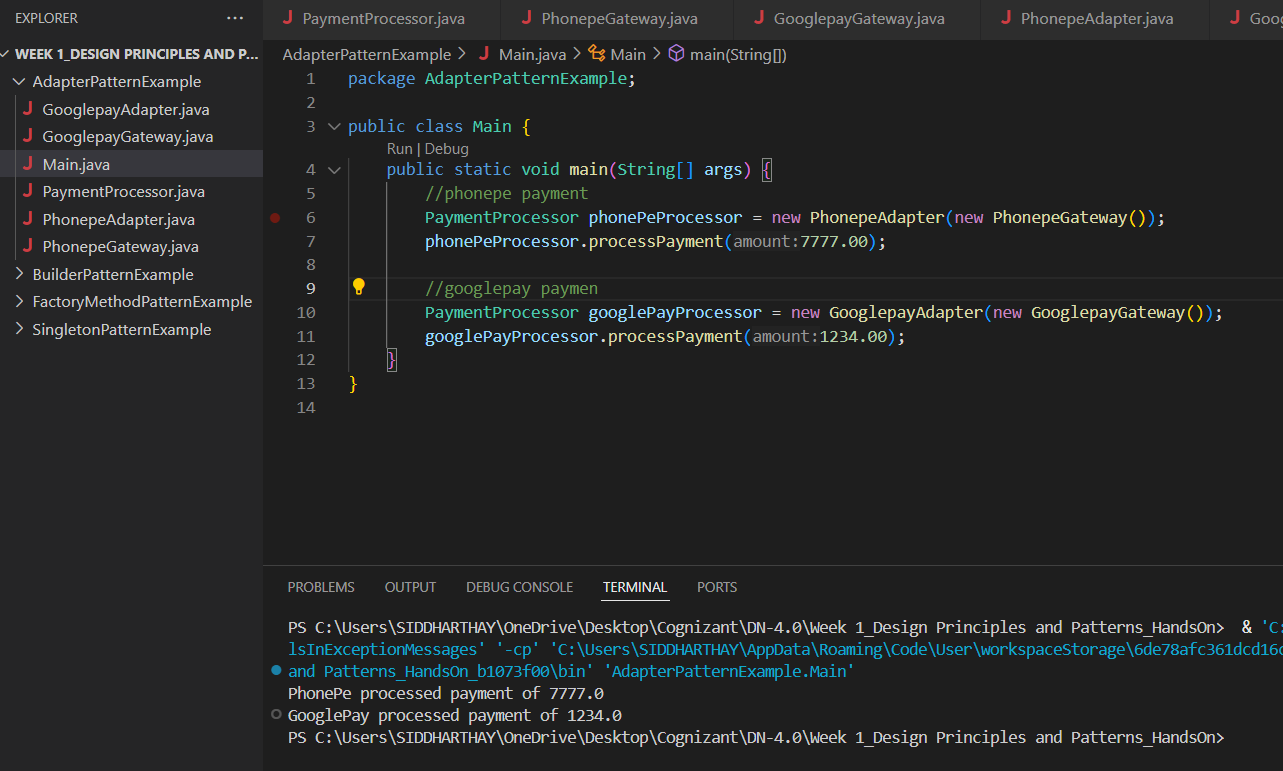
        PaymentProcessor googlePayProcessor = new GooglepayAdapter(new GooglepayGateway());

        googlePayProcessor.processPayment(1234.00);

    }

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**CODE:**

//Notifier.java

public interface Notifier {

    void send(String message);

}

//NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    @Override

    public void send(String message) {

        notifier.send(message);

    }

}

//EmailNotifier.java

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

        System.out.println("Sending Email Message: " + message);

    }

}

//SlackNotifierDecorator.java

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSlack(message);

    }

    private void sendSlack(String message) {

        System.out.println("Sending Slack Message: " + message);

    }

}

//SMSNotifierDecorator.java

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message); //first sending existing notifier message

        sendSMS(message);    //then add sms functionality

    }

    private void sendSMS(String message) {

        System.out.println("Sending SMS Message: " + message);

    }

}

//Main.java

public class Main {

    public static void main(String[] args) {

        //email

        Notifier emailNotifier = new EmailNotifier();

        //sms

        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

        //slack

        Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

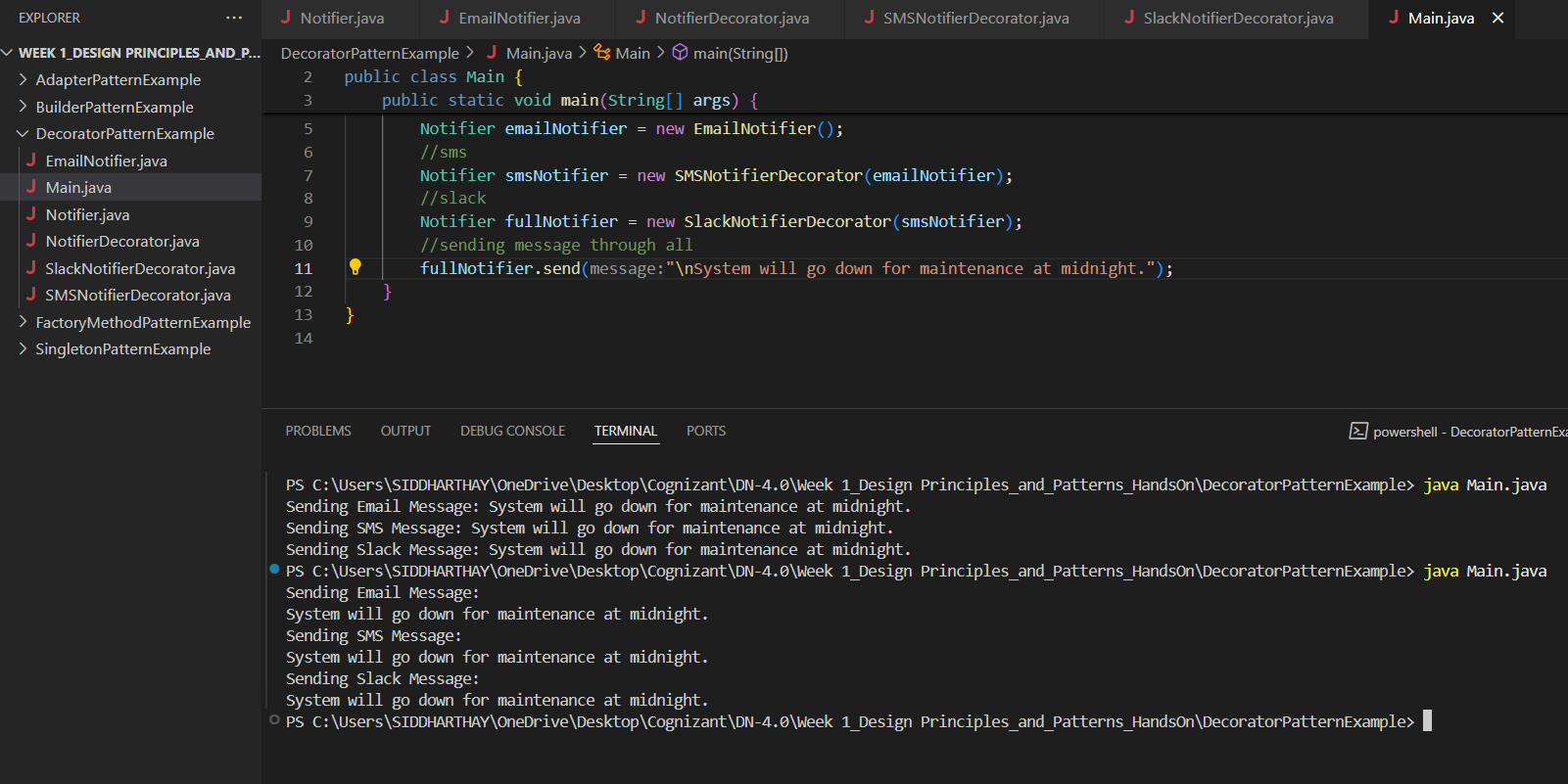
        //sending message through all

        fullNotifier.send("\nSystem will go down for maintenance at midnight.");

    }

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

//Image.java

public interface Image {

    void display();

}

//RealImage.java

public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName) {

        this.fileName = fileName;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

        System.out.println("Loading image: " + fileName);

    }

    @Override

    public void display() {

        System.out.println("Displaying image: " + fileName);

    }

}

//ProxyImage.java

public class ProxyImage implements Image {

    private RealImage realImage;

    private String fileName;

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(fileName);

        }

        realImage.display();

    }

}

//Main.java

public class Main {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("my\_photo.jpg");

        System.out.println("First call to display():");

        image1.display();  //load from remote and display

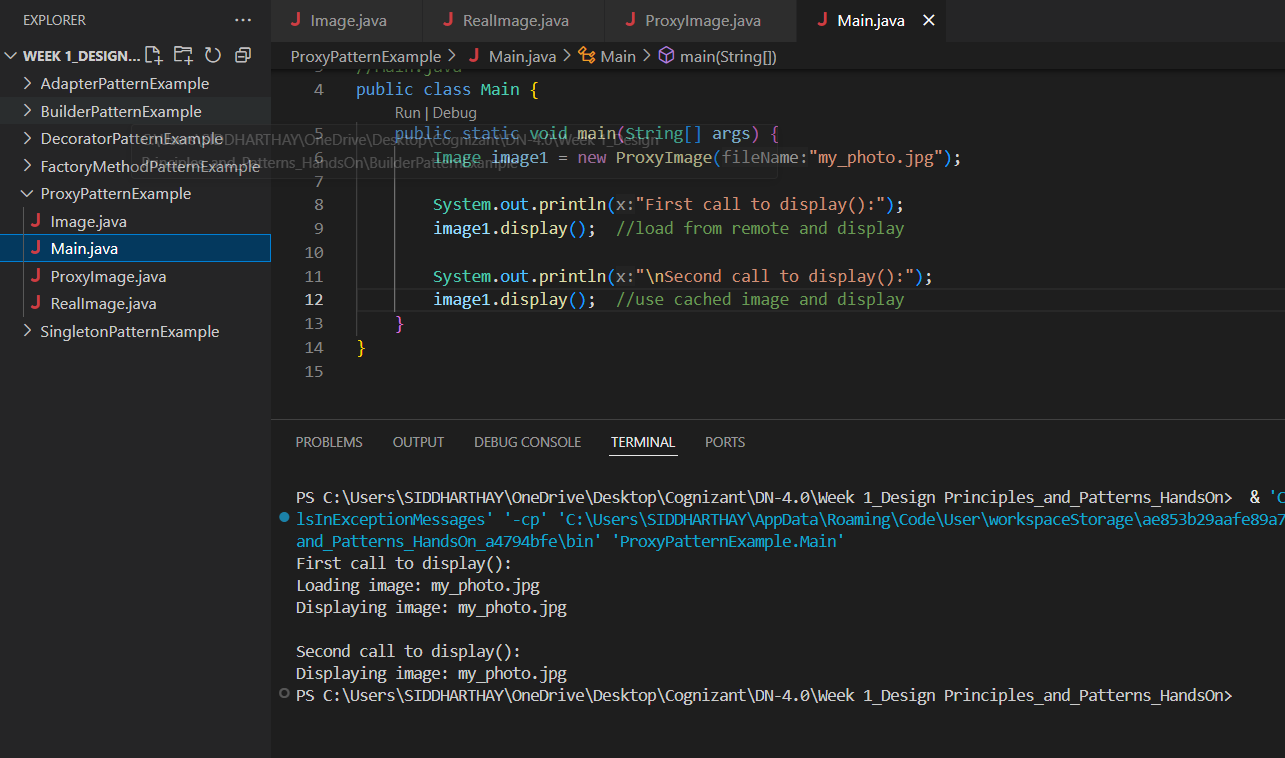
        System.out.println("\nSecond call to display():");

        image1.display();  //use cached image and display

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**CODE:**

//Stock.java

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

//StockMarket.java

import java.util.\*;

public class StockMarket implements Stock {

    private List<Observer> observers = new ArrayList<>();

    private String stockName;

    private double stockPrice;

    public void setStock(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        notifyObservers();  //notifing when stock is updated

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

//MobileApp.java

public class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + appName + "] Stock Update -> " + stockName + ": INR " + stockPrice);

    }

}

//WebApp.java

public class WebApp implements Observer {

    private String webAppName;

    public WebApp(String webAppName) {

        this.webAppName = webAppName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + webAppName + "] Stock Update -> " + stockName + ": INR " + stockPrice);

    }

}

//Observer.java

public interface Observer {

    void update(String stockName, double stockPrice);

}

//Main.java

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("StockMobile");

        Observer webApp = new WebApp("StockWeb");

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        stockMarket.setStock("CTSH", 6493.73);

        stockMarket.setStock("INFY", 1420.50);

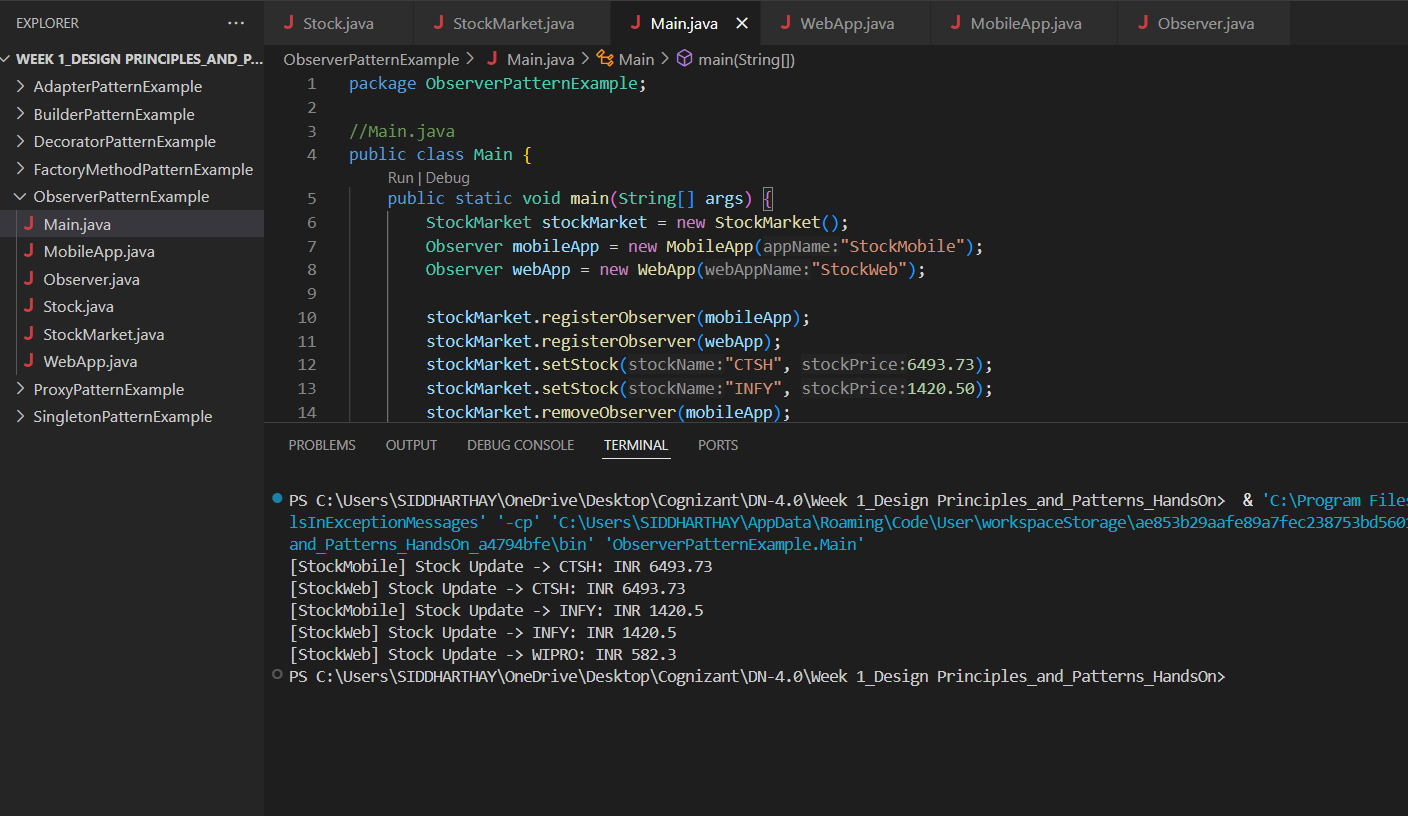
        stockMarket.removeObserver(mobileApp);

        stockMarket.setStock("WIPRO", 582.30);

    }

}

**OUPUT:**

****